UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/720,726	11/24/2003	Adam G. Wolff	LZLO-01006US0	2756

28554 7590 10/14/2008

VIERRA MAGEN MARCUS & DENIRO LLP
575 MARKET STREET SUITE 2500
SAN FRANCISCO, CA 94105

EXAMINER	
CHEN, QING	

ART UNIT	PAPER NUMBER
2191	

MAIL DATE	DELIVERY MODE
10/14/2008	PAPER

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

Office Action Summary	Application No.	Applicant(s)	
	10/720,726	WOLFF ET AL.	
	Examiner	Art Unit	
	Qing Chen	2191	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on *01 August 2008*.

2a)☐ This action is **FINAL**. 2b)☒ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1-15,17,19-31,33-35,38-58 and 60-65* is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) *1-15,17,19-31,33-35,38-58 and 60-65* is/are rejected.

7)☐ Claim(s) _____ is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☒ The specification is objected to by the Examiner.

10)☒ The drawing(s) filed on *01 August 2008* is/are: a)☒ accepted or b)☐ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☐ All b)☐ Some * c)☐ None of:

1.☐ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____.

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

1) ☐ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☐ Information Disclosure Statement(s) (PTO/SB/08) Paper No(s)/Mail Date _____.
4) ☐ Interview Summary (PTO-413) Paper No(s)/Mail Date. _____.
5) ☐ Notice of Informal Patent Application
6) ☐ Other: _____.

DETAILED ACTION

1. This Office action is in response to the RCE filed on August 1, 2008.

2. **Claims 1-15, 17, 19-31, 33-35, 38-58, and 60-65** are pending.

3. **Claims 1-3, 6-8, 10, 11, 13, 17, 19, 20, 33, 34, 38, 40, 48, 49, 52, 58, and 62** have been amended.

4. **Claims 16, 18, 32, 36, 37, and 59** have been cancelled.

5. The objection to the drawings is withdrawn in view of Applicant's submission of the replacement drawing sheets.

6. The objections to the specification due to ineffective attempt to incorporate subject matter and informalities are withdrawn in view of Applicant's amendments to the specification. However, Applicant's amendments to the specification fail to address the objection due to the use of trademarks. Accordingly, this objection is maintained and further explained hereinafter.

7. The objections to Claims 16 and 40 are withdrawn in view of Applicant's amendments to the claims or cancellation of the claims.

8. The 35 U.S.C. § 112, second paragraph, rejections of Claims 1-17, 38-58, and 60-65 are withdrawn in view of Applicant's amendments to the claims or cancellation of the claims.

9. It is noted that Claims 1, 33, 38, 52, and 62 contain amendments that are submitted without markings to indicate the changes that have been made relative to the immediate prior version of the claim.

10. It is noted that Claim 3 contains an amendment by adding a period (.) at the end of the claim. However, the immediate prior version of Claim 3 already contains a period (.) at the end of the claim.

11. It is noted that Claim 20 contains an amendment by deleting a period (.) at the end of the second-to-last limitation. However, the text of any deleted subject matter must be shown by being placed within double brackets, not single brackets. See 37 CFR 1.121(c)(2).

Continued Examination Under 37 CFR 1.114

12. A request for continued examination under 37 CFR 1.114, including the fee set forth in 37 CFR 1.17(e), was filed in this application after final rejection. Since this application is eligible for continued examination under 37 CFR 1.114, and the fee set forth in 37 CFR 1.17(e) has been timely paid, the finality of the previous Office action has been withdrawn pursuant to 37 CFR 1.114. Applicant's submission filed on August 1, 2008 has been entered.

Response to Amendment

Specification

13. The use of trademarks, such as WINDOWS and JAVASCRIPT, has been noted in this application—namely, in paragraphs [0037] and [0068] of the specification. Trademarks should be capitalized wherever they appear (capitalize each letter OR accompany each trademark with an appropriate designation symbol, *e.g.*, ™ or ®) and be accompanied by the generic terminology (use trademarks as adjectives modifying a descriptive noun, *e.g.*, "the JAVA programming language").

Although the use of trademarks is permissible in patent applications, the proprietary nature of the marks should be respected and every effort made to prevent their use in any manner, which might adversely affect their validity as trademarks.

14. Applicant submitted a preliminary amendment to the specification (received on 04/16/2004) which, among other amendments, deleted paragraph [0027] of the specification. Thus, the paragraph numbering of the amended specification (as amended by the preliminary amendment) is different than the paragraph numbering of the originally-filed specification, beginning with paragraph [0027] (*i.e.*, original paragraph [0028] is now paragraph [0027], original paragraph [0029] is now paragraph [0028], and so forth). The preliminary amendment was entered and considered. In response to the objections to the specification in the Non-Final Rejection (mailed on 06/27/2007), the Applicant submitted an amendment to the specification (received on 12/26/2007) correcting the various informalities. However, some of the instructions specifying the paragraph numbers of the amended specification to be replaced with the paragraph numbers of the originally-filed specification were incorrect. Examiner pointed out these typographical errors in paragraph 12 of the Final Rejection (mailed on 03/03/2008). Applicant's response to the Final Rejection (mailed on 03/03/2008) did not address these typographical errors. Applicant is advised to submit a substitute specification, excluding the claims, pursuant to 37 CFR 1.125 in order to avoid any unnecessary confusion with regard to the paragraph numbering.

Claim Objections

15. **Claims 20-35, 38-57, and 62-65** are objected to because of the following informalities:

- **Claim 20** contains the following typographical errors:

- The word "and" after the "initializing the first object code file [...]" limitation should be deleted.
- "[R]elation between objects" should read -- relations between objects --.

- **Claims 20 and 33** recite the limitation "the runtime." Applicant is advised to change this limitation to read "the runtime environment" for the purpose of providing it with proper explicit antecedent basis.
- **Claims 21-31** depend on Claim 20 and, therefore, suffer the same deficiency as Claim 20.
- **Claims 34 and 35** depend on Claim 33 and, therefore, suffer the same deficiency as Claim 33.
- **Claim 38** recites the limitations "said processor readable storage devices" and "the application." Applicant is advised to change these limitations to read "said one or more processor readable storage devices" and "the computer application," respectively, for the purpose of providing them with proper explicit antecedent bases.
- **Claims 39-51** depend on Claim 38 and, therefore, suffer the same deficiency as Claim 38.
- **Claim 52** recites the limitation "the runtime environment." Applicant is advised to change this limitation to read "the first runtime environment" for the purpose of providing it with proper explicit antecedent basis.
- **Claims 53-57** depend on Claim 52 and, therefore, suffer the same deficiency as Claim 52.

- **Claim 62** recites the limitation "a second state." Applicant is advised to change this limitation to read "a second application state" for the purpose of keeping the claim language consistent throughout the claims.
- **Claims 63-65** depend on Claim 62 and, therefore, suffer the same deficiency as Claim 62.

Appropriate correction is required.

Claim Rejections - 35 USC § 112

16. The following is a quotation of the second paragraph of 35 U.S.C. 112:

> The specification shall conclude with one or more claims particularly pointing out and distinctly claiming the subject matter which the applicant regards as his invention.

17. **Claims 20-31** are rejected under 35 U.S.C. 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention.

Claim 20 recites the limitation "the application objects." There is insufficient antecedent basis for this limitation in the claim. In the interest of compact prosecution, the Examiner subsequently interprets this limitation as reading "application objects" for the purpose of further examination.

Claims 21-31 depend on Claim 20 and, therefore, suffer the same deficiency as Claim 20.

Claim Rejections - 35 USC § 102

18. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (e) the invention was described in (1) an application for patent, published under section 122(b), by another filed in the United States before the invention by the applicant for patent or (2) a patent granted on an application for patent by another filed in the United States before the invention by the applicant for patent, except that an international application filed under the treaty defined in section 351(a) shall have the effects for purposes of this subsection of an application filed in the United States only if the international application designated the United States and was published under Article 21(2) of such treaty in the English language.

19. **Claims 1, 2, 4, 6-8, 11-15, 17, 19-23, 25, 27-31, 33, 38, 39, 41, 42, 44, 45, 47-58, and 60-62** are rejected under 35 U.S.C. 102(e) as being anticipated by **US 7,191,441 (hereinafter "Abbott")**.

As per **Claim 1**, Abbott discloses:

- creating a serialized representation of application objects in the runtime environment *(see Column 9: 10-20, "... the preferred embodiment of the present invention provides a set of callable routines and tools that allow a user to store away the state of an initialised Java application. The stored application is available for subsequent reloading, but without the start-up cost of initialisation. In other words, a "snapshot" can effectively be taken of an application at some user-defined point in time and saved for later use."; Column 16: 37-54, "There are a number of run-specific variables and data structures used by the components and sub-*

components of the Java VM such as the garbage collector or JIT, which need to be saved if, on reload, the Java VM is to function correctly." and "The first captures the component's current state and places it in a data structure which it will return to the calling save routine.");

- building an optimized object code file using the serialized representation and the first object code file, wherein the optimized object code file includes instructions creating relations between objects in the runtime environment *(see Figure 7; Column 11: 61-64, "In the preferred embodiment, the addresses in the heap are translated at save time to a chain of reference addresses and then back again at restore. A vector of object references is utilised, the entries of which are initialised to 0 (null)."; Column 17: 32-60, "FIG. 7 is a flowchart summarising the above save operation. Thus processing starts with the application calling the save routine (step 705) ..." and "We are now ready to start the save operation proper, and this commences with saving first the class information (step 720), and then the heap (step 725), both as described above." and "Finally the method concludes with calling the JVM system components to perform their individual save operations (step 745), before the save routine eventually terminates (step 750) with the completed snapshot.")*; and

- loading the optimized object code file into to a new runtime environment to create a second application state isomorphic to the first application state *(see Column 3: 57-60, "In addition, the snapshot can be transmitted over a network, thereby allowing resumption on another system, for example for diagnostic purposes."; Column 18: 10-13, "In the preferred embodiment, the saved state is loaded by a special "javaload" tool, which loads the stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM.")*.

As per **Claim 2**, the rejection of **Claim 1** is incorporated; and Abbott further discloses:

- reading from a runtime memory space a description of each object of a running computer application *(see Column 12: 6-10, "Thus the first item is to create a vector or table 510 having three columns. Each object in turn in the heap is then identified from the system Reference queue, and added into the table 510, with one object entry per row of the table.").*

As per **Claim 4**, the rejection of **Claim 2** is incorporated; and Abbott further discloses:

- wherein the runtime environment is a virtual machine *(see Figure 2: 40).*

As per **Claim 6**, the rejection of **Claim 1** is incorporated; and Abbott further discloses:

- identifying each object of a running computer application by a unique identifier *(see Column 12: 10-15, "The first column 511 is used to hold the identifier or reference number of an object, and the third column 513 is used to hold the length of the object. The actual contents of the object can now be copied over into the save file for the snapshot (not shown in FIG. 5).").*

As per **Claim 7**, the rejection of **Claim 6** is incorporated; and Abbott further discloses:

- detaching each object from an object hierarchy and creating a description of each slot in said object *(see Column 12: 29-36, "Therefore, as shown in the sequence of diagrams FIG. 5A, 5B, and 5C, the heap is scanned to detect all inter-object references. In the example of FIG. 5A, two such references are present, one 532A from object A to object S, and one 533A from*

object F to object S. As each object reference is found, a linked list is built up for that object, which is headed from the second column 512 of the vector table 510.").

As per **Claim 8**, the rejection of **Claim 6** is incorporated; and Abbott further discloses:

- providing the serialized representation directly to the building step *(see Column 18: 10-14, "... the saved state is loaded by a special "javaload" tool, which loads the stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM. The reloading of the application is performed in stages.").*

As per **Claim 11**, the rejection of **Claim 1** is incorporated; and Abbott further discloses:

- assigning a serialization identifier to each object *(see Column 11: 46-50, "All instance objects can be found from the Reference Queue of the Java VM. In turn each object may contain references to other objects. The problem with these "object to object" references is that they refer to absolute locations (serialization identifier) inside the heap.").*

As per **Claim 12**, the rejection of **Claim 1** is incorporated; and Abbott further discloses:

- developing the computer application in an interpreted language *(see Column 7: 28-33, "Another component of the Java VM is the interpreter 156, which is responsible for reading in Java byte code from loaded classes, and converting this into machine instructions for the relevant platform.").*

As per **Claim 13**, the rejection of **Claim 6** is incorporated; and Abbott further discloses:

- assigning a function ID to each function in the computer application *(see Column 16: 9-15, "... a table is created containing: the name of the DLL (function ID); its load location (memory address); and the NativeLibrary class association (in the case of DLLs containing JNI code).").*

As per **Claim 14**, the rejection of **Claim 13** is incorporated; and Abbott further discloses:

- creating a function ID table associating each function ID with function code *(see Column 16: 9-15, "... a table (function ID table) is created containing: the name of the DLL (function ID); its load location (memory address); and the NativeLibrary class association (in the case of DLLs containing JNI code).").*

As per **Claim 15**, the rejection of **Claim 13** is incorporated; and Abbott further discloses:

- assigning unique function identifiers to functions within closures *(see Column 16: 2-7, "In order to save the state associated with such DLLs, all loaded DLLs are recorded with their base addresses (function identifiers), and if they contain JNI code, all references to them from class loaders and NativeLibrary classes (functions within closures) are also recorded (NativeLibrary classes provide the mechanism for a Java application to access the DLLs).").*

As per **Claim 17**, the rejection of **Claim 1** is incorporated; and Abbott further discloses:

- writing the serialized representation to a text file prior to said step of building *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization)." and 55-57, "A*

particular benefit of this approach is where multiple, identical, VMs can all be rapidly launched from a single save file (text file).").

As per **Claim 19**, the rejection of **Claim 1** is incorporated; and Abbott further discloses:

- wherein the step of creating a serialized representation is performed in a different runtime *(see Column 9: 55-58, "Thus the method commences with the start of the Java application whose state is to be saved (step 410), which in turn leads in standard fashion to the initialisation of the Java VM classes (step 420)." Note that the runtime whose state is to be saved is the old runtime environment (different runtime).).*

As per **Claim 20**, Abbott discloses:

- compiling an application provided in a source language to create a first object code file *(see Column 11: 32 and 33, "The JIT-compiled code for the method needs to be located and analysed to detect usage of absolute memory references." and 38-40, "In a preferred implementation, the user can request that the JIT is called to compile all non JIT-compiled methods immediately before the snapshot is taken.")*;

- initializing the first object code file in a runtime environment to create a first application state *(see Column 11: 34-36, "Relevant information from these addresses is then stored with the JIT-compiled code, so that the addresses can be properly adjusted when code is loaded back into memory.")*;

- creating a serialized representation of application objects in the runtime environment *(see Column 9: 10-20, "... the preferred embodiment of the present invention provides a set of*

callable routines and tools that allow a user to store away the state of an initialised Java application. The stored application is available for subsequent reloading, but without the start-up cost of initialisation. In other words, a "snapshot" can effectively be taken of an application at some user-defined point in time and saved for later use."; Column 16: 37-54, "There are a number of run-specific variables and data structures used by the components and sub-components of the Java VM such as the garbage collector or JIT, which need to be saved if, on reload, the Java VM is to function correctly." and "The first captures the component's current state and places it in a data structure which it will return to the calling save routine."); and

- building an optimized object code file using the serialized representation and the first object code file, wherein the optimized object code file includes instructions creating relations between objects in the runtime environment to create a second application state isomorphic to the first application state *(see Figure 7; Column 11: 61-64, "In the preferred embodiment, the addresses in the heap are translated at save time to a chain of reference addresses and then back again at restore. A vector of object references is utilised, the entries of which are initialised to 0 (null)."; Column 17: 32-60, "FIG. 7 is a flowchart summarising the above save operation. Thus processing starts with the application calling the save routine (step 705) ..." and "We are now ready to start the save operation proper, and this commences with saving first the class information (step 720), and then the heap (step 725), both as described above." and "Finally the method concludes with calling the JVM system components to perform their individual save operations (step 745), before the save routine eventually terminates (step 750) with the completed snapshot.").*

As per **Claim 21**, the rejection of **Claim 20** is incorporated; and Abbott further discloses:

- reading from the runtime environment a description of each object of the application *(see Column 12: 6-10, "Thus the first item is to create a vector or table 510 having three columns. Each object in turn in the heap is then identified from the system Reference queue, and added into the table 510, with one object entry per row of the table.").*

As per **Claim 22**, the rejection of **Claim 21** is incorporated; and Abbott further discloses:

- outputting the description to a rebuilder *(see Column 18: 10-14, "... the saved state is loaded by a special "javaload" tool, which loads the stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM. The reloading of the application is performed in stages.").*

As per **Claim 23**, the rejection of **Claim 22** is incorporated; and Abbott further discloses:

- storing the serialized representation in a text file and providing the text file to the rebuilder *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization)." and 55-57, "A particular benefit of this approach is where multiple, identical, VMs can all be rapidly launched from a single save file (text file)."; Column 18: 10-14, "... the saved state is loaded by a special "javaload" tool, which loads the stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM. The reloading of the application is performed in stages.").*

As per **Claim 25**, the rejection of **Claim 20** is incorporated; and Abbott further discloses:

- wherein the runtime environment is a virtual machine *(see Figure 2: 40)*.

As per **Claim 27**, the rejection of **Claim 20** is incorporated; and Abbott further discloses:

- assigning a serialization identifier to each initialized object *(see Column 11: 46-50, "All instance objects can be found from the Reference Queue of the Java VM. In turn each object may contain references to other objects. The problem with these "object to object" references is that they refer to absolute locations (serialization identifier) inside the heap.")*.

As per **Claim 28**, the rejection of **Claim 20** is incorporated; and Abbott further discloses:

- enumerating each object in a global scope and writing a serialized description of each said object *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization)."; Column 12: 6-10, "Thus the first item is to create a vector or table 510 having three columns. Each object in turn in the heap is then identified from the system Reference queue (enumerating each object), and added into the table 510, with one object entry per row of the table.")*.

As per **Claim 29**, the rejection of **Claim 20** is incorporated; and Abbott further discloses:

- assigning a function ID to each function in the application *(see Column 16: 9-15, "... a table is created containing: the name of the DLL (function ID); its load location (memory address); and the NativeLibrary class association (in the case of DLLs containing JNI code).")*.

As per **Claim 30**, the rejection of **Claim 29** is incorporated; and Abbott further discloses:

- creating a function ID table associating each function ID with a function call *(see Column 16: 9-15, "... a table (function ID table) is created containing: the name of the DLL (function ID); its load location (memory address); and the NativeLibrary class association (in the case of DLLs containing JNI code).").*

As per **Claim 31**, the rejection of **Claim 29** is incorporated; and Abbott further discloses:

- assigning function identifiers to functions within closures *(see Column 16: 2-7, "In order to save the state associated with such DLLs, all loaded DLLs are recorded with their base addresses (function identifiers), and if they contain JNI code, all references to them from class loaders and NativeLibrary classes (functions within closures) are also recorded (NativeLibrary classes provide the mechanism for a Java application to access the DLLs).").*

As per **Claim 33**, Abbott discloses:

- requesting an application from an application source server *(see Column 3: 57-60, "In addition, the snapshot can be transmitted over a network, thereby allowing resumption on another system, for example for diagnostic purposes."; Column 19: 56-58, "For example, the invention has been described primarily in relation to Java in a server environment ...")*;

- receiving a first object code file loaded into a runtime environment and creating a first application state *(see Column 11: 34-36, "Relevant information from these addresses is then*

stored with the JIT-compiled code, so that the addresses can be properly adjusted when code is loaded back into memory.");

- receiving an optimized object code file using a serialized description of the application from the application source server and the first object code file, the optimized object code file including instructions creating relations between objects in the runtime environment *(see Figure 7; Column 11: 61-64, "In the preferred embodiment, the addresses in the heap are translated at save time to a chain of reference addresses and then back again at restore. A vector of object references is utilised, the entries of which are initialised to 0 (null)."; Column 17: 32-60, "FIG. 7 is a flowchart summarising the above save operation. Thus processing starts with the application calling the save routine (step 705) ..." and "We are now ready to start the save operation proper, and this commences with saving first the class information (step 720), and then the heap (step 725), both as described above." and "Finally the method concludes with calling the JVM system components to perform their individual save operations (step 745), before the save routine eventually terminates (step 750) with the completed snapshot.")*; and

- loading the optimized object code file into a new runtime environment to create a second application state isomorphic to the first application state *(see Column 3: 57-60, "In addition, the snapshot can be transmitted over a network, thereby allowing resumption on another system, for example for diagnostic purposes."; Column 18: 10-13, "In the preferred embodiment, the saved state is loaded by a special "javaload" tool, which loads the stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM.")*.

As per **Claim 38**, Abbott discloses:

- creating a serialized representation of application objects in the runtime environment *(see Column 9: 10-20, "... the preferred embodiment of the present invention provides a set of callable routines and tools that allow a user to store away the state of an initialised Java application. The stored application is available for subsequent reloading, but without the start-up cost of initialisation. In other words, a "snapshot" can effectively be taken of an application at some user-defined point in time and saved for later use."; Column 16: 37-54, "There are a number of run-specific variables and data structures used by the components and sub-components of the Java VM such as the garbage collector or JIT, which need to be saved if, on reload, the Java VM is to function correctly." and "The first captures the component's current state and places it in a data structure which it will return to the calling save routine.")*;

- building an optimized object code file using the serialized representation of the application objects and the first object code file, wherein the optimized object code file includes instructions creating relations between objects in the runtime environment *(see Figure 7; Column 11: 61-64, "In the preferred embodiment, the addresses in the heap are translated at save time to a chain of reference addresses and then back again at restore. A vector of object references is utilised, the entries of which are initialised to 0 (null)."; Column 17: 32-60, "FIG. 7 is a flowchart summarising the above save operation. Thus processing starts with the application calling the save routine (step 705) ..." and "We are now ready to start the save operation proper, and this commences with saving first the class information (step 720), and then the heap (step 725), both as described above." and "Finally the method concludes with calling the JVM*

system components to perform their individual save operations (step 745), before the save routine eventually terminates (step 750) with the completed snapshot."); and

- loading the optimized object code file into a new runtime environment to create a second application state isomorphic to the first application state *(see Column 3: 57-60, "In addition, the snapshot can be transmitted over a network, thereby allowing resumption on another system, for example for diagnostic purposes."; Column 18: 10-13, "In the preferred embodiment, the saved state is loaded by a special "javaload" tool, which loads the stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM.").*

As per **Claim 39**, the rejection of **Claim 38** is incorporated; and Abbott further discloses:

- reading from a runtime environment memory space a description of each object of a running application *(see Column 12: 6-10, "Thus the first item is to create a vector or table 510 having three columns. Each object in turn in the heap is then identified from the system Reference queue, and added into the table 510, with one object entry per row of the table.").*

As per **Claim 41**, the rejection of **Claim 38** is incorporated; and Abbott further discloses:

- identifying each object of a running application by a unique identifier *(see Column 12: 10-15, "The first column 511 is used to hold the identifier or reference number of an object, and the third column 513 is used to hold the length of the object. The actual contents of the object can now be copied over into the save file for the snapshot (not shown in FIG. 5).").*

As per **Claim 42**, the rejection of **Claim 41** is incorporated; and Abbott further discloses:

- writing a description to a text file and compiling the text file *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization)." and 55-57, "A particular benefit of this approach is where multiple, identical, VMs can all be rapidly launched from a single save file (text file)."; Column 4: 21-26, "The preferred embodiment provides the user with various options, as to whether certain actions should be performed prior to determining the current state of the components of the virtual machine. Examples of such actions include performing a garbage collection, and forcing compilation of at least some of the application (compiling the text file).").*

As per **Claim 44**, the rejection of **Claim 41** is incorporated; and Abbott further discloses:

- detaching each object from an object hierarchy and creating a description of each slot in said object *(see Column 12: 29-36, "Therefore, as shown in the sequence of diagrams FIG. 5A, 5B, and 5C, the heap is scanned to detect all inter-object references. In the example of FIG. 5A, two such references are present, one 532A from object A to object S, and one 533A from object F to object S. As each object reference is found, a linked list is built up for that object, which is headed from the second column 512 of the vector table 510.").*

As per **Claim 45**, the rejection of **Claim 41** is incorporated; and Abbott further discloses:

- determining whether the object is a class *(see Column 11: 5-8, "... all loaded classes are held as objects 145 on the heap 140, and may contain absolute references to instance objects, which need to be encoded for storage.")*; and

- writing a serialized description of the class *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization).").*

As per **Claim 47**, the rejection of **Claim 39** is incorporated; and Abbott further discloses:

- assigning a serialization identifier to each object *(see Column 11: 46-50, "All instance objects can be found from the Reference Queue of the Java VM. In turn each object may contain references to other objects. The problem with these "object to object" references is that they refer to absolute locations (serialization identifier) inside the heap.").*

As per **Claim 48**, the rejection of **Claim 39** is incorporated; and Abbott further discloses:

- developing the computer application in an interpreted language *(see Column 7: 28-33, "Another component of the Java VM is the interpreter 156, which is responsible for reading in Java byte code from loaded classes, and converting this into machine instructions for the relevant platform.").*

As per **Claim 49**, the rejection of **Claim 41** is incorporated; and Abbott further discloses:

- assigning a function ID to each function in the computer application *(see Column 16: 9-15, "... a table is created containing: the name of the DLL (function ID); its load location*

(memory address); and the NativeLibrary class association (in the case of DLLs containing JNI code).").

As per **Claim 50**, the rejection of **Claim 49** is incorporated; and Abbott further discloses:

- creating a function ID table associating each function ID with function code *(see Column 16: 9-15, "... a table (function ID table) is created containing: the name of the DLL (function ID); its load location (memory address); and the NativeLibrary class association (in the case of DLLs containing JNI code).").*

As per **Claim 51**, the rejection of **Claim 49** is incorporated; and Abbott further discloses:

- assigning function identifiers to functions within closures *(see Column 16: 2-7, "In order to save the state associated with such DLLs, all loaded DLLs are recorded with their base addresses (function identifiers), and if they contain JNI code, all references to them from class loaders and NativeLibrary classes (functions within closures) are also recorded (NativeLibrary classes provide the mechanism for a Java application to access the DLLs).").*

As per **Claim 52**, Abbott discloses:

- compiling the application into a first object code file *(see Column 11: 32 and 33, "The JIT-compiled code for the method needs to be located and analysed to detect usage of absolute memory references." and 38-40, "In a preferred implementation, the user can request that the JIT is called to compile all non JIT-compiled methods immediately before the snapshot is taken.");*

- loading the first object code file into a first runtime environment to create a first application state *(see Column 11: 34-36, "Relevant information from these addresses is then stored with the JIT-compiled code, so that the addresses can be properly adjusted when code is loaded back into memory.")*;

- creating a serialized representation of a memory space in said first runtime environment *(see Column 9: 10-20, "... the preferred embodiment of the present invention provides a set of callable routines and tools that allow a user to store away the state of an initialised Java application. The stored application is available for subsequent reloading, but without the start-up cost of initialisation. In other words, a "snapshot" can effectively be taken of an application at some user-defined point in time and saved for later use."; Column 16: 37-54, "There are a number of run-specific variables and data structures used by the components and sub-components of the Java VM such as the garbage collector or JIT, which need to be saved if, on reload, the Java VM is to function correctly." and "The first captures the component's current state and places it in a data structure which it will return to the calling save routine.")*;

- building a second object code file using said serialized representation and the first object code file, wherein the second object code file includes instructions creating relations between objects in the first runtime environment *(see Figure 7; Column 11: 61-64, "In the preferred embodiment, the addresses in the heap are translated at save time to a chain of reference addresses and then back again at restore. A vector of object references is utilised, the entries of which are initialised to 0 (null)."; Column 17: 32-60, "FIG. 7 is a flowchart summarising the above save operation. Thus processing starts with the application calling the save routine (step 705) ..." and "We are now ready to start the save operation proper, and this*

commences with saving first the class information (step 720), and then the heap (step 725), both as described above." and "Finally the method concludes with calling the JVM system components to perform their individual save operations (step 745), before the save routine eventually terminates (step 750) with the completed snapshot."); and

- loading said second object code into a second runtime environment to create a second application state isomorphic to the first application state *(see Column 3: 57-60, "In addition, the snapshot can be transmitted over a network, thereby allowing resumption on another system, for example for diagnostic purposes."; Column 18: 10-13, "In the preferred embodiment, the saved state is loaded by a special "javaload" tool, which loads the stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM.").*

As per **Claim 53**, the rejection of **Claim 52** is incorporated; and Abbott further discloses:

- wherein the step of compiling is performed on an interpreted language application *(see Column 7: 28-33, "Another component of the Java VM is the interpreter 156, which is responsible for reading in Java byte code from loaded classes, and converting this into machine instructions for the relevant platform.").*

As per **Claim 54**, the rejection of **Claim 52** is incorporated; and Abbott further discloses:

- wherein the step of creating is performed by calling at least one function from said first runtime environment *(see Column 9: 55-58, "Thus the method commences with the start of the Java application whose state is to be saved (step 410), which in turn leads in standard fashion to the initialisation of the Java VM classes (step 420).").*

As per **Claim 55**, the rejection of **Claim 52** is incorporated; and Abbott further discloses:

- wherein the step of creating is performed in the same runtime environment as said application *(see Column 9: 55-58, "Thus the method commences with the start of the Java application whose state is to be saved (step 410), which in turn leads in standard fashion to the initialisation of the Java VM classes (step 420).").*

As per **Claim 56**, the rejection of **Claim 52** is incorporated; and Abbott further discloses:

- wherein the step of creating is performed in a different runtime environment from said application *(see Column 9: 55-58, "Thus the method commences with the start of the Java application whose state is to be saved (step 410), which in turn leads in standard fashion to the initialisation of the Java VM classes (step 420)." Note that the runtime whose state is to be saved is the old runtime environment (different runtime).).*

As per **Claim 57**, the rejection of **Claim 52** is incorporated; and Abbott further discloses:

- executing portions of the application marked for execution prior to said creating step *(see Column 10: 16-21, "... it may be desirable to perform some relatively trivial operation on the classes, such as accessing a class variable, prior to the save step (step 440), in order to ensure that the generic initialisations are indeed completed before the save operation.").*

As per **Claim 58**, Abbott discloses:

- receiving from a runtime environment a serialized representation of objects in a memory space of the runtime environment *(see Column 9: 10-20, "... the preferred embodiment of the present invention provides a set of callable routines and tools that allow a user to store away the state of an initialised Java application. The stored application is available for subsequent reloading, but without the start-up cost of initialisation. In other words, a "snapshot" can effectively be taken of an application at some user-defined point in time and saved for later use."; Column 16: 37-54, "There are a number of run-specific variables and data structures used by the components and sub-components of the Java VM such as the garbage collector or JIT, which need to be saved if, on reload, the Java VM is to function correctly." and "The first captures the component's current state and places it in a data structure which it will return to the calling save routine.")*; and

- building an optimized object code file using the serialized representation and a compiled object code file used to create the memory space, wherein the optimized object code file includes instructions creating relations between objects in the runtime environment *(see Figure 7; Column 11: 61-64, "In the preferred embodiment, the addresses in the heap are translated at save time to a chain of reference addresses and then back again at restore. A vector of object references is utilised, the entries of which are initialised to 0 (null)."; Column 17: 32-60, "FIG. 7 is a flowchart summarising the above save operation. Thus processing starts with the application calling the save routine (step 705) ..." and "We are now ready to start the save operation proper, and this commences with saving first the class information (step 720), and then the heap (step 725), both as described above." and "Finally the method concludes with*

calling the JVM system components to perform their individual save operations (step 745), before the save routine eventually terminates (step 750) with the completed snapshot.").

As per **Claim 60**, the rejection of **Claim 58** is incorporated; and Abbott further discloses:

- initializing a serialization process in a separate memory space to create said serialized representation *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization).").*

As per **Claim 61**, the rejection of **Claim 58** is incorporated; and Abbott further discloses:

- initializing a serialization process in the runtime environment to create said serialized representation *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization).").*

As per **Claim 62**, Abbott discloses:

- creating a serialized representation of application objects in the runtime environment *(see Column 9: 10-20, "... the preferred embodiment of the present invention provides a set of callable routines and tools that allow a user to store away the state of an initialised Java application. The stored application is available for subsequent reloading, but without the start-up cost of initialisation. In other words, a "snapshot" can effectively be taken of an application at some user-defined point in time and saved for later use."; Column 16: 37-54, "There are a*

number of run-specific variables and data structures used by the components and sub-components of the Java VM such as the garbage collector or JIT, which need to be saved if, on reload, the Java VM is to function correctly." and "The first captures the component's current state and places it in a data structure which it will return to the calling save routine.");

- building an optimized object code file using the serialized representation and the first object code file, wherein the optimized object code file includes instructions creating relations between objects in the runtime environment *(see Figure 7; Column 11: 61-64, "In the preferred embodiment, the addresses in the heap are translated at save time to a chain of reference addresses and then back again at restore. A vector of object references is utilised, the entries of which are initialised to 0 (null)."; Column 17: 32-60, "FIG. 7 is a flowchart summarising the above save operation. Thus processing starts with the application calling the save routine (step 705) ..." and "We are now ready to start the save operation proper, and this commences with saving first the class information (step 720), and then the heap (step 725), both as described above." and "Finally the method concludes with calling the JVM system components to perform their individual save operations (step 745), before the save routine eventually terminates (step 750) with the completed snapshot.")*; and

- loading the optimized object code file into a new runtime environment via the network to create a second application state isomorphic to the first application state *(see Column 3: 57-60, "In addition, the snapshot can be transmitted over a network, thereby allowing resumption on another system, for example for diagnostic purposes."; Column 18: 10-13, "In the preferred embodiment, the saved state is loaded by a special "javaload" tool, which loads the*

stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM.").

Claim Rejections - 35 USC § 103

20. The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

21. **Claims 3, 40, 63, and 64** are rejected under 35 U.S.C. 103(a) as being unpatentable over **Abbott** in view of **US 2004/0044989 (hereinafter "Vachuska")**.

As per **Claim 3**, the rejection of **Claim 1** is incorporated; however, Abbott does not disclose:

- enumerating a description of each object of the computer application using reflection.

Vachuska discloses:

- enumerating a description of each object of the computer application using reflection *(see Paragraph [0024], "The package 'java.lang.reflect' provides classes and interfaces for obtaining reflective information about classes and objects.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Vachuska into the teaching of Abbott to include enumerating a description of each object of the computer application using reflection.

The modification would be obvious because one of ordinary skill in the art would be motivated to make it possible to examine the structure of the object *(see Vachuska – Paragraph [0004])*.

As per **Claim 40**, the rejection of **Claim 38** is incorporated; however, Abbott does not disclose:

- enumerating a description of each object of the computer application using reflection.

Vachuska discloses:

- enumerating a description of each object of the computer application using reflection *(see Paragraph [0024], "The package 'java.lang.reflect' provides classes and interfaces for obtaining reflective information about classes and objects.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Vachuska into the teaching of Abbott to include enumerating a description of each object of the computer application using reflection. The modification would be obvious because one of ordinary skill in the art would be motivated to make it possible to examine the structure of the object *(see Vachuska – Paragraph [0004])*.

As per **Claim 63**, the rejection of **Claim 62** is incorporated; however, Abbott does not disclose:

- enumerating a description of each object of the application using reflection.

Vachuska discloses:

- enumerating a description of each object of the application using reflection *(see Paragraph [0024], "The package 'java.lang.reflect' provides classes and interfaces for obtaining reflective information about classes and objects.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Vachuska into the teaching of Abbott to include enumerating a description of each object of the application using reflection. The modification would be obvious because one of ordinary skill in the art would be motivated to make it possible to examine the structure of the object *(see Vachuska – Paragraph [0004]).*

As per **Claim 64**, the rejection of **Claim 63** is incorporated; and Abbott further discloses:

- wherein the new runtime environment is a virtual machine *(see Figure 2: 40).*

22. **Claims 5, 9, 10, 24, 26, 35, 43, and 46** are rejected under 35 U.S.C. 103(a) as being unpatentable over **Abbott** in view of **US 2003/0195923 (hereinafter "Bloch")**.

As per **Claim 5**, the rejection of **Claim 4** is incorporated; however, Abbott does not disclose:

- wherein the virtual machine is a presentation renderer.

Bloch discloses:

- wherein the virtual machine is a presentation renderer *(see Paragraph [0032], "In one embodiment, client Presentation Renderer 14 is a Macromedia Flash Player embedded in a web client as a plug-in.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Bloch into the teaching of Abbott to include wherein the virtual machine is a presentation renderer. The modification would be obvious because one of ordinary skill in the art would be motivated to access Internet applications and entertainment *(see Bloch – Paragraph [0006])*.

As per **Claim 9**, the rejection of **Claim 1** is incorporated; however, Abbott does not disclose:

- wherein the serialized representation of application objects in a runtime environment is written in an Extensible Markup Language data format.

Bloch discloses:

- wherein the serialized representation of application objects in a runtime environment is written in an Extensible Markup Language data format *(see Paragraph [0030], "In one environment, the mark-up language description is an XML-based language that is designed specifically for describing an application's user interface, along with the connection of that user-interface to various data sources and/or web services.")*.

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Bloch into the teaching of Abbott to include wherein the serialized representation of application objects in a runtime environment is written in an Extensible Markup Language data format. The modification would be obvious because one of ordinary skill in the art would be motivated to facilitate the sharing of data across different information systems, particularly via the Internet.

As per **Claim 10**, the rejection of **Claim 9** is incorporated; and Abbott further discloses:

- storing a file prior to said step of building *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization).").*

However, Abbott does not disclose:

- a markup language file.

Bloch discloses:

- a markup language file *(see Paragraph [0030], "In one environment, the mark-up language description is an XML-based language that is designed specifically for describing an application's user interface, along with the connection of that user-interface to various data sources and/or web services.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Bloch into the teaching of Abbott to include a markup language file. The modification would be obvious because one of ordinary skill in the art would be motivated to facilitate the sharing of data across different information systems, particularly via the Internet.

As per **Claim 24**, the rejection of **Claim 22** is incorporated; and Abbott further discloses:

- writing the description to a file and providing the file to the rebuilder *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization)."; Column 18: 10-14, "... the*

saved state is loaded by a special "javaload" tool, which loads the stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM. The reloading of the application is performed in stages.").

However, Abbott does not disclose:

- an Extensible Markup Language file.

Bloch discloses:

- an Extensible Markup Language file *(see Paragraph [0030], "In one environment, the mark-up language description is an XML-based language that is designed specifically for describing an application's user interface, along with the connection of that user-interface to various data sources and/or web services.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Bloch into the teaching of Abbott to include an Extensible Markup Language file. The modification would be obvious because one of ordinary skill in the art would be motivated to facilitate the sharing of data across different information systems, particularly via the Internet.

As per **Claim 26**, the rejection of **Claim 25** is incorporated; however, Abbott does not disclose:

- wherein the virtual machine is a presentation renderer.

Bloch discloses:

- wherein the virtual machine is a presentation renderer *(see Paragraph [0032], "In one embodiment, client Presentation Renderer 14 is a Macromedia Flash Player embedded in a web client as a plug-in.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Bloch into the teaching of Abbott to include wherein the virtual machine is a presentation renderer. The modification would be obvious because one of ordinary skill in the art would be motivated to access Internet applications and entertainment *(see Bloch – Paragraph [0006]).*

As per **Claim 35**, the rejection of **Claim 33** is incorporated; however, Abbott does not disclose:

- wherein the runtime memory state is of a presentation renderer.

Bloch discloses:

- wherein the runtime memory state is of a presentation renderer *(see Paragraph [0032], "In one embodiment, client Presentation Renderer 14 is a Macromedia Flash Player embedded in a web client as a plug-in.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Bloch into the teaching of Abbott to include wherein the runtime memory state is of a presentation renderer. The modification would be obvious because one of ordinary skill in the art would be motivated to access Internet applications and entertainment *(see Bloch – Paragraph [0006]).*

As per **Claim 43**, the rejection of **Claim 41** is incorporated; and Abbott further discloses:

- writing a description to a file and compiling a file *(see Column 3: 18-20, "The principle underlying the present invention is serialisation, whereby a "snapshot" is taken of the VM state, which is then stored in a file (serialization)."; Column 4: 21-26, "The preferred embodiment provides the user with various options, as to whether certain actions should be performed prior to determining the current state of the components of the virtual machine. Examples of such actions include performing a garbage collection, and forcing compilation of at least some of the application.").*

However, Abbott does not disclose:

- an Extensible Markup Language file and a markup language file.

Bloch discloses:

- an Extensible Markup Language file and a markup language file *(see Paragraph [0030], "In one environment, the mark-up language description is an XML-based language that is designed specifically for describing an application's user interface, along with the connection of that user-interface to various data sources and/or web services.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Bloch into the teaching of Abbott to include an Extensible Markup Language file and a markup language file. The modification would be obvious because one of ordinary skill in the art would be motivated to facilitate the sharing of data across different information systems, particularly via the Internet.

As per **Claim 46**, the rejection of **Claim 39** is incorporated; however, Abbott does not disclose:

- wherein the serialized representation of application objects in a runtime environment is written in an Extensible Markup Language data format.

Bloch discloses:

- wherein the serialized representation of application objects in a runtime environment is written in an Extensible Markup Language data format *(see Paragraph [0030], "In one environment, the mark-up language description is an XML-based language that is designed specifically for describing an application's user interface, along with the connection of that user-interface to various data sources and/or web services.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Bloch into the teaching of Abbott to include wherein the serialized representation of application objects in a runtime environment is written in an Extensible Markup Language data format. The modification would be obvious because one of ordinary skill in the art would be motivated to facilitate the sharing of data across different information systems, particularly via the Internet.

23. **Claim 34** is rejected under 35 U.S.C. 103(a) as being unpatentable over **Abbott** in view of **US 5,987,256 (hereinafter "Wu")**.

As per **Claim 34**, the rejection of **Claim 33** is incorporated; however, Abbott does not disclose:

- wherein the object code includes media assets.

Wu discloses:

- wherein the object code includes media assets *(see Column 19: 35-37, "... a standard object file, such as an HTML or JAVA image, is input online 1300 to a compiler 1301 which runs on a standard computer 1302.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Wu into the teaching of Abbott to include wherein the object code includes media assets. The modification would be obvious because one of ordinary skill in the art would be motivated to execute various media contents.

24. **Claim 65** is rejected under 35 U.S.C. 103(a) as being unpatentable over **Abbott** in view of **Vachuska** as applied to Claim 64 above, and further in view of **Bloch**.

As per **Claim 65**, the rejection of **Claim 64** is incorporated; however, Abbott and Vachuska do not disclose:

- wherein the virtual machine is a presentation renderer.

Bloch discloses:

- wherein the virtual machine is a presentation renderer *(see Paragraph [0032], "In one embodiment, client Presentation Renderer 14 is a Macromedia Flash Player embedded in a web client as a plug-in.").*

Therefore, it would have been obvious to one of ordinary skill in the art at the time the invention was made to incorporate the teaching of Bloch into the teaching of Abbott to include

wherein the virtual machine is a presentation renderer. The modification would be obvious because one of ordinary skill in the art would be motivated to access Internet applications and entertainment *(see Bloch – Paragraph [0006])*.

Response to Arguments

25. Applicant's arguments with respect to Claims 1, 20, 33, 38, 52, 58, and 62 have been considered but are moot in view of the new ground(s) of rejection.

In the Remarks, Applicant argues:

a) Abbott does not disclose "building an optimized object code file using the serialized representation and the first object code file, wherein the optimized object code file includes instructions creating relations between objects in the runtime to create a second application state isomorphic to the first application state."

Examiner's response:

a) Examiner disagrees. With respect to the Applicant's assertion that Abbott does not disclose "building an optimized object code file using the serialized representation and the first object code file, wherein the optimized object code file includes instructions creating relations between objects in the runtime environment to create a second application state isomorphic to the first application state," the Examiner respectfully submits that Abbott clearly discloses "building an optimized object code file using the serialized representation and the first object code file, wherein the optimized object code file includes instructions creating relations between objects in

the runtime environment to create a second application state isomorphic to the first application state" *(see Figure 7; Column 11: 61-64, "In the preferred embodiment, the addresses in the heap are translated at save time to a chain of reference addresses and then back again at restore. A vector of object references is utilised, the entries of which are initialised to 0 (null)."; Column 17: 32-60, "FIG. 7 is a flowchart summarising the above save operation. Thus processing starts with the application calling the save routine (step 705) ..." and "We are now ready to start the save operation proper, and this commences with saving first the class information (step 720), and then the heap (step 725), both as described above." and "Finally the method concludes with calling the JVM system components to perform their individual save operations (step 745), before the save routine eventually terminates (step 750) with the completed snapshot.")*. Note that the completed snapshot (optimized object code file) is generated by combining all of the saved state information of a Java application which includes the JIT-compiled code for the class methods (the first object code file) and the system components of the Java VM (the serialized representation). The completed snapshot also contains state information for all instance objects by maintaining a vector of object references (instructions creating relations between objects). Further note that once the saved state is loaded in another Java VM, the suspended Java application is ready to run. Thus, one of ordinary skill in the art would readily comprehend that the state of the Java application after the snapshot is loaded (the second application state) is the same as the state of the Java application prior to the snapshot is saved (the first application state). The Java application is suspended in one Java VM and subsequently resumed in another Java VM. In other words, the state of the Java application running in the first Java VM is preserved in

the second Java VM and thus, the state of the Java application remains isomorphic between the two runtime environments.

In the Remarks, Applicant argues:

b) Neither Abbott nor Laney, alone or in combination, disclose "building an optimized object code file using the serialized representation and the first object code file" and "loading the optimized object code file into a new runtime environment to create a second application state isomorphic to the first application state."

In contrast to the method recited in claim 1, both Abbott and Laney require a special "loader program" to restore the application. Abbott discloses that

> the saved state [of the application] is loaded by a special 'javaload' tool, which loads the stored application into a Java VM, bypassing all the initialization normally associated with bringing up a Java VM. The reloading of the application is performed in stages. [emphasis added] Col. 18, lines 10- 13.

Fig. 8 of Abbott illustrates the steps executed by the "special 'javaload' tool" to restore the application. In contrast, the method recited in claim 1 builds an "optimized object code using the serialized representation and the first object code file." The method recited in claim 1 does not require a special "loader tool" to restore the application. The "optimized object code" is simply executed by the "new runtime environment."

Examiner's response:

b) Examiner disagrees. Applicant's arguments are not persuasive for at least the following reasons:

First, with respect to the Applicant's assertion that Abbott does not disclose "building an optimized object code file using the serialized representation and the first object code file," the Examiner has addressed the Applicant's arguments regarding Abbott does not disclose "building an optimized object code file using the serialized representation and the first object code file" in the Examiner's response (a) hereinabove.

Second, without acquiesce to the Applicant's assertion that Abbott does not disclose "loading the optimized object code file into a new runtime environment to create a second application state isomorphic to the first application state," the Examiner first submits that the claim language does not require executing the optimized object code in the new runtime environment as averred by the Applicant. The claims recite, exactly, "loading the optimized object code file into a new runtime environment" (emphasis added). Thus, the claims are not limited to the scope of executing the optimized object code in the new runtime environment. Applicant is reminded that in order for such limitations to be considered, the claim language requires to specifically recite such limitations in the claims, otherwise broadest reasonable interpretations of the broadly claimed limitations are deemed to be proper.

Third, with respect to the Applicant's assertion that Abbott does not disclose "loading the optimized object code file into a new runtime environment to create a second application state isomorphic to the first application state," the Examiner respectfully submits that Abbott clearly discloses "loading the optimized object code file into a new runtime environment to create a second application state isomorphic to the first application state" *(see Column 3: 57-60, "In addition, the snapshot can be transmitted over a network, thereby allowing resumption on another system, for example for diagnostic purposes."; Column 18: 10-13, "In the preferred*

embodiment, the saved state is loaded by a special "javaload" tool, which loads the stored application into a Java VM, bypassing all the initialisation normally associated with bringing up a Java VM."). Note that the "javaload" tool restores the Java application by loading the saved state into a new Java VM. Examiner has addressed the Applicant's arguments regarding Abbott does not disclose "creat[ing] a second application state isomorphic to the first application state" in the Examiner's response (a) hereinabove.

Conclusion

26. Any inquiry concerning this communication or earlier communications from the Examiner should be directed to Qing Chen whose telephone number is 571-270-1071. The Examiner can normally be reached on Monday through Thursday from 7:30 AM to 4:00 PM. The Examiner can also be reached on alternate Fridays.

If attempts to reach the Examiner by telephone are unsuccessful, the Examiner's supervisor, Wei Zhen, can be reached on 571-272-3708. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Any inquiry of a general nature or relating to the status of this application or proceeding should be directed to the TC 2100 Group receptionist whose telephone number is 571-272-2100.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR

system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).

/Q. C./

Examiner, Art Unit 2191

/Wei Y Zhen/

Supervisory Patent Examiner, Art Unit 2191